Reducción y Recursión

Taller de Álgebra I

Segundo cuatrimestre de 2015

Reducción

- En el contexto de los lenguajes funcionales, llamamos modelo de cómputo al modo en que se calcula el valor de una expresión.
- El mecanismo de evaluación en Haskell es la reducción:
  1. Reemplazamos una subexpresión por otra.
  2. La subexpresión a reemplazar es alguna instancia del lado izquierdo de alguna ecuación orientada del programa, y se la llama radical o redex (*reducible expression*).
  3. La reemplazaremos por el lado derecho de esa misma ecuación, instanciado de manera acorde.
  4. El resto de la expresión no cambia.

Reducción

Dado el siguiente programa:

```
resta :: Integer -> Integer -> Integer
resta x y = x - y

suma :: Integer -> Integer -> Integer
suma x y = x + y

digitos :: Integer -> Integer
digitos x = ??
```

- Qué sucede al evaluar `suma (resta 2 (digitos 42)) 4`
- Buscamos un redex y una asignación: $\texttt{suma}\ (\underbrace{\texttt{resta 2 (digitos 42)}}_{\text{redex}})\ \texttt{4}$
  - $\texttt{x} \leftarrow \texttt{2}$
  - $\texttt{y} \leftarrow \texttt{(digitos 42)}$
- Reemplazamos el redex con esa asignación:
  $\texttt{suma (resta 2 (digitos 42)) 4} \rightsquigarrow \texttt{suma (2 - (digitos 42)) 4}$

Formas normales

- Las expresiones se reducen hasta que no haya más redexes.
- Como resultado se obtiene una forma normal (que no tiene un cómputo asociado).
- **Mecanismo de reducción:**
  1. Si la expresión está en forma normal, terminamos.
  2. Si no, buscar un redex, reemplazarlo y volver a empezar.

Confluencia de estrategias de reducción

- ¿Toda expresión tiene forma normal? **¡No!**
  1. `f x = f (f x)` – ¿cuánto vale `f 3`?
  2. `infinito = infinito + 1` – ¿cuánto vale `infinito`?
  3. `inverso x | x /= 0 = 1 / x` – ¿cuánto vale `inverso 0`?
- Cuando existe una forma normal, ¿es única? **¡Sí!**
  - Esta propiedad se llama confluencia.

Indefinición

- Las expresiones que no tienen forma normal se dicen que están indefinidas ($\perp$).
- ¿Podemos definir en Haskell una función que siempre se indefine?
- En el preludio de Haskell se define como:
  `undefined :: a`
- Cualquier intento de evaluar `undefined` se indefine.
  `g :: Integer -> Integer`
  `g x | x == undefined = 1`
  `g 2` $\rightsquigarrow$ `undefined` – Este 'reduce a' no quiere decir haya un *redex*, sino que el resultado es el mismo.

Indefinición

¿Cómo podemos clasificar las funciones?

- Funciones totales: nunca se indefinen, nunca pueden ser equivalentes a `undefined`

```
suc :: Integer -> Integer
suc x = x + 1
```

- Funciones parciales: hay argumentos para los cuales se indefinen, pueden ser equivalentes a `undefined`

```
inv :: Float -> Float
inv x | x /= 0 = 1/x
```

Evaluación estricta vs. no estricta

¿Qué sucede si intentamos aplicar una función sobre una expresión que se indefine?

- Evaluación estricta: siempre '`f undefined` $\rightsquigarrow$ `undefined`' – Nuevamente, este 'reduce a' no quiere decir haya un *redex*, sino que el resultado es el mismo.
- Evaluación no estricta: puede pasar '`f undefined` $\rightsquigarrow$ [algun valor]'.

```
const ::  a -> b -> a
const x y = x
```

¿A qué expresión reduce '`const 2 undefined`'?
¡Depende del **diseño** del lenguaje!
El secreto está en el orden de evaluación.

Órdenes de evaluación

Orden aplicativo o eager (“ansioso”):

Empieza por reducir los redexes internos y continúa hacia afuera; es decir que primero evalúa los argumentos y después la función.

Ejemplo:

```
 suma (3+4) (suc (2*3))
⇝ suma (3+4) (suc 6)
⇝ suma 7 (suc 6)
⇝ suma 7 (6 + 1)
⇝ suma 7 7
⇝ 7 + 7
⇝ 14
```

Órdenes de evaluación

Orden normal o lazy ("perezoso"):

Reduce el redex más externo para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero evalúa la función y después los argumentos (si se necesitan).

(mismo) Ejemplo:

```
 suma (3+4) (suc (2*3))
⇝  (3+4) + (suc (2*3))
⇝  7 + (suc (2*3))
⇝  7 + ((2*3) + 1)
⇝  7 + (6 + 1)
⇝  7 + 7
⇝  14
```

Órdenes de evaluación

Orden aplicativo o eager:

- Primero redexes internos.
- Primero los argumentos, después la función.

Orden normal o lazy:

- El redex más externo para el que pueda saber qué ecuación del programa se debe aplicar.
- Primero la función, después los argumentos (si se necesitan).

Observaciones:

- En caso de haber más de un redex en el mismo nivel, ambas estrategias proceden de izquierda a derecha.
- El orden 'lazy' siempre encuentra la forma normal, cuando existe.

Evaluación en Haskell

Consiste en aplicar el orden ‘lazy’ (redexes externos primero).

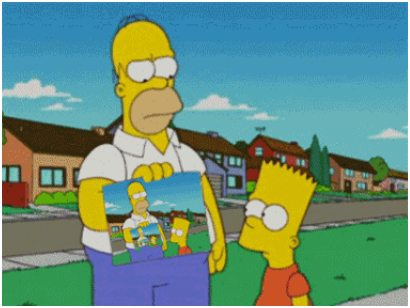
Recursión

- Hasta ahora, especificamos funciones que consistían en "expresiones sencillas".
- Sin embargo, el tipo de funciones que hicimos hasta la clase pasada no permite programar todas las funciones computables.
- Por ejemplo, ¿cómo es una función en Haskell para calcular el factorial de un número entero?

$$n! = \prod_{k=1}^{n} k, \qquad n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si no} \end{cases}$$

¡La segunda definición de `factorial` involucra a esta misma función del lado derecho!

Definiciones recursivas



```
factorial :: Integer -> Integer
factorial n | n == 0 = 1
factorial n | n > 0 = n * factorial (n-1)
```

Definiciones recursivas

- Propiedades de una definición recursiva:
  1. Tiene que tener uno o más casos base.
  2. Las llamadas recursivas del lado derecho tienen que *acercarse* al caso base, con relación a los parámetros del lado izquierdo de la ecuación.
- En cierto sentido, la recursión es el equivalente computacional de la inducción para las demostraciones.

Otras funciones recursivas

Programar las siguientes funciones

- Implementar la función `fib :: Integer -> Integer` que devuelve el i-ésimo número de Fibonacci. Recordar que la secuencia de Fibonacci se define como:

$$fib(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fib(n-1) + fib(n-2) & \text{en otro caso} \end{cases}$$

- Implementar la función `par :: Integer -> Bool` que determine si un número es par. No está permitido utilizar mod ni div.
- Implementar la función `sumaImpares :: Integer -> Integer` que dado $n \in \mathbb{N}$ sume los primeros n números impares. Ej: `sumaImpares 3` $\rightsquigarrow$ `1+3+5` $\rightsquigarrow$ `9`.
- Escribir una función para determinar si un número es múltiplo de 3. No está permitido utilizar mod ni div.

Asegurarse de llegar a un caso base

- Consideremos este programa recursivo para determinar si un número es par:
- ```
  par ::  Integer -> Bool
  par 0 = True
  par n = par (n-2)
  ```

  ¿Qué problema tiene esta función?

  ¿Cómo se arregla?
- ```
  par 0 = True
  par 1 = False
  par n = par (n-2)
  ```
- ```
  par 0 = True
  par n = not (par (n-1))
  ```
  ```

Ejercicios

1. Escribir una función `doblefact` para calcular $n!! = n\,(n-2)(n-4)\ldots 2$.
   Por ejemplo: `doblefact` $10 \rightsquigarrow 10 * 8 * 6 * 4 * 2 \rightsquigarrow 3840$.
   La función se debe indefinir para los números impares.
2. Escribir una función que dados n, $m \in \mathbb{N}$ compute el combinatorio $\binom{n}{m}$.
   Hacerlo usando la igualdad $\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$.
3. Escribir una función recursiva que no termine si se la ejecuta con números negativos (y en cambio sí termine para el resto de los números).
4. Escribir una función que dado $n \in \mathbb{N}$ sume los números impares positivos cuyo cuadrado sea menor que n. Por ejemplo:
   `sumaImparesCuyoCuadSeaMenorQue` $30 \rightsquigarrow 1 + 3 + 5 \rightsquigarrow 9$.